Science of Computer Programming 5 (1985) 97-106
North-Holland

SMALL PROGRAMMING EXERCISES 6

M. REM

Department of Mathematics and Computing Science, Eindhoven University of Technology, 5600 MB Eindhoven, The Netherlands

As announced in the preceding issue, we continue our graph computations. Many computations on graphs deal with the existence of paths of one kind or another. Such exercises may be formulated in a number of ways, for example: determine all vertices to which there is a path from a given vertex, determine whether there exists a path between two given vertices, record such paths, etc. Usually one formulation does not make the exercise more complicated than another. In our graph problems each of these types of formulations occurs at least once.

We have three new exercises, all dealing with directed graphs. Each of these may be solved by the tripartitioning technique explained in the preceding issue. The last exercise allows a solution that is linear in the number of vertices and arcs. The other two, in which the arcs have weights attached to them, require the selection of a particular grey vertex to be coloured black. This makes the computation times of their solutions worse than linear.

Exercise 14: ascending paths

Every arc in a directed graph G has a given weight: the arc from vertex j to vertex $e(i)$, $b(j) \leq i < b(j+1)$, has weight $d(i)$. An *ascending path* is a path in which the weight of every arc is at least the maximum of the weights of all preceding arcs of that path. We are requested to determine all vertices that can be reached via an ascending path from a given vertex X, i.e. we have to solve S in

$$
\begin{array}{l}
|[N, M, X: \mathit{int}\ \{N \geq 1 \wedge M \geq 0 \wedge 0 \leq X < N\} \\
\quad ; b(j: 0 \leq j \leq N), e, d(i: 0 \leq i < M): \textbf{array of } \mathit{int} \\
\qquad \{\mathrm{SUC}(G, b, e)\} \\
\quad ; |[r(j: 0 \leq j < N): \textbf{array of } \mathit{bool} \\
\qquad ; S \\
\qquad \{(\mathbf{A}j: 0 \leq j < N: r(j) \equiv (G \text{ has an ascending path from } X \text{ to } j))\} \\
\quad]| \\
]|
\end{array}
$$

Exercise 15: the tunnels

X and Y are two points in a given network of (one-way) tunnels. Each tunnel has a given positive height. We have to determine the maximum height of a motortruck that has to drive from X to Y.

We state the exercise more formally in terms of graphs. A directed graph is called *strongly connected* if for every vertex j and for every vertex k there exists a path from j to k. As in Exercise 14 the arcs of graph G have weights. In this case G is strongly connected and the weights are positive. The *weight of a path* is defined to be the minimum of the weights of its arcs. We are requested to compute the maximum of the weights of the paths from a given vertex X to a given vertex Y:

$$
\begin{array}{l}
|[N, M, X, Y: \mathit{int}\ \{N \geqslant 1 \wedge M \geqslant 0 \wedge 0 \leqslant X < N \wedge 0 \leqslant Y < N\} \\
\quad ; b(j: 0 \leqslant j \leqslant N), e, d(i: 0 \leqslant i < M): \textbf{array of } \mathit{int} \\
\qquad \{\mathrm{SUC}(G, b, e) \wedge G \text{ strongly connected} \wedge (\mathbf{A}i: 0 \leqslant i < M: d(i) \geqslant 1)\} \\
\quad ; |[z: \mathit{int} \\
\qquad ; S \\
\qquad \{z = (\text{maximum of the weights of the paths from } X \text{ to } Y)\} \\
\quad]| \\
]|
\end{array}
$$

Exercise 16: two-person game

We consider a two-person game. The players take turns in making moves until a position is reached from which no move can be made. The one who is to move then has lost. We have to determine for each position whether it is won for the one who has to make a move, lost, or undecided.

The rules of the game are given by a directed graph, the vertices representing the positions and the arcs the possible moves. A *strategy* s is a mapping from the subset of vertices that have at least one successor to the set of all vertices, such that $s(j)$ is a successor of j for all j in the domain of s. Let j be a vertex and s and t strategies. The (s, t)*-path from* j is the path

$$j, s(j), t(s(j)), s(t(s(j))), \ldots$$

If this path is finite it ends in a vertex without successors.

Vertex j is called *won* if

$$
\begin{array}{l}
(\mathbf{E}s{:}s \text{ a strategy: } (\mathbf{A}t{:}\ t \text{ a strategy:} \\
\qquad \text{the } (s, t)\text{-path from } j \text{ is finite and has odd length}))
\end{array}
$$

(Remember that, since the length of a path is its number of arcs, a path of odd length has an even number of vertices.) Vertex j is called *lost* if

$$
\begin{array}{l}
(\mathbf{A}s{:}\ s \text{ a strategy: } (\mathbf{E}t{:}\ t \text{ a strategy:} \\
\qquad \text{the } (s, t)\text{-path from } j \text{ is finite and has even length}))
\end{array}
$$

The conditions for won and lost, obviously, exclude each other. All other vertices are called *undecided.*

The graph is given by listing for each vertex the set of its predecessors, rather than—as in Exercises 14 and 15—that of its successors. We have to find a statement

list S such that

$$
\begin{array}{l}
|[N, M: \mathit{int}\ \{N \geq 1 \wedge M \geq 0\} \\
; b(j: 0 \leq j \leq N), e(i: 0 \leq i < M): \textbf{array of } \mathit{int} \\
\quad \{\text{PRED}(G, b, e)\} \\
; |[q(j: 0 \leq j < N): \textbf{array of } \mathit{int} \\
\quad ; S \\
\qquad \{(\mathbf{A}j: 0 \leq j < N: \quad (q(j) = -1) \equiv (\text{vertex } j \text{ is lost}) \\
\qquad\qquad\qquad\qquad \wedge (q(j) = 0) \equiv (\text{vertex } j \text{ is undecided}) \\
\qquad\qquad\qquad\qquad \wedge (q(j) = 1) \equiv (\text{vertex } j \text{ is won}))\} \\
\quad]| \\
]|
\end{array}
$$

Solution of Exercise 11 (acyclicity)

We are requested to find a statement list S such that

$$
\begin{array}{l}
|[N, M: \mathit{int}\ \{N \geq 1 \wedge M \geq 0\} \\
; b(j: 0 \leq j \leq N), e(i: 0 \leq i < M): \textbf{array of } \mathit{int} \\
\quad \{\text{SUC}(G, b, e)\} \\
; |[ac: \mathit{bool} \\
\quad ; S \\
\qquad \{ac \equiv (G \text{ is acyclic})\} \\
\quad]| \\
]|
\end{array}
$$

For V a set of vertices, $SG(V)$ denotes the subgraph of G that is induced by V, i.e. the graph having V as its set of vertices and an arc between two vertices whenever that arc is in G. It seems attractive to have an invariant of the form

$$(SG(V) \text{ acyclic}) \equiv (G \text{ acyclic})$$

This may be initialized by making V equal to the set of vertices of G. We wish to delete vertices from V. If $j \in V$ has no predecessors in $SG(V)$ it does not belong to a cycle in $SG(V)$ and, consequently,

$$(SG(V \backslash \{j\}) \text{ acyclic}) \equiv (SG(V) \text{ acyclic})$$

Remark. The same is, of course, true for vertices without successors. Deleting those vertices from V would lead to a program that requires the recording of the predecessors of each vertex.

For W a set of vertices and $j \in W$, we denote by $P(j, W)$ the set of predecessors of j in $SG(W)$ and by $S(j, W)$ the set of successors of j in $SG(W)$. If in P or S the argument W is dropped the set of vertices of G is meant. The subset of V consisting of all vertices j for which $P(j, V) = \emptyset$ is the set of candidates for deletion

from V. Calling this subset $V1$ and the remainder of V set $V0$ we obtain as our invariant $P0 \wedge P1$:

$P0$: $(SG(V0 \cup V1)\text{ acyclic}) \equiv (G\text{ acyclic})$

$P1$: $(\mathbf{A}j: j \in V0: P(j, V0 \cup V1) \neq \emptyset)$
$\wedge (\mathbf{A}j: j \in V1: P(j, V0 \cup V1) = \emptyset)$

Since the only vertices that are deleted from $V0 \cup V1$ are those without predecessors in $V0 \cup V1$, we also maintain invariant $P2$:

$P2$: $(\mathbf{A}j: j \in V0 \cup V1: S(j) \subseteq V0)$

Upon the deletion of a vertex j from $V1$ we have for each $k \in S(j, V0 \cup V1)$ that $j \in P(k, V0 \cup V1)$ and, consequently, that $P(k, V0 \cup V1)$ shrinks by one element. This requires that each such k for which $P(k, V0 \cup V1)$ has become empty is moved from $V0$—to which it belongs on account of $P2$—to $V1$. Notice, moreover, that because of $P2$ we have for $j \in V1$ that $S(j, V0 \cup V1) = S(j)$. The repetition terminates with $V1 = \emptyset$. This yields with $P0$

$$(SG(V0)\text{ acyclic}) \equiv (G\text{ acyclic})$$

From $V1 = \emptyset \wedge P1$ we conclude that

$$(\mathbf{A}j: j \in V0: P(j, V0) \neq \emptyset)$$

i.e. that every vertex in $SG(V0)$ has a predecessor. $SG(V0)$ is then acyclic only if $V0 = \emptyset$.

We thus arrive at a program of the following structure:

```
  V0, V1 := {j | P(j) ≠ ∅}, {j | P(j) = ∅}
; do V1 ≠ ∅
  → let j ∈ V1
    ; V1 := V1\{j}
    ; for each k ∈ S(j)
      if P(k, V0 ∪ V1) = ∅ → V0, V1 := V0\{k}, V1 ∪ {k}
      □ P(k, V0 ∪ V1) ≠ ∅ → skip
      fi
  od
; ac := (V0 = ∅)
```

In order to be able to express the guards of the selection above in terms of program variables, we introduce an integer array $t(j: 0 \leq j < N)$ and maintain as an invariant

$$(\mathbf{A}k: k \in V0 \cup V1: t(k) = (\mathbf{N}j: j \in V0 \cup V1: j \in P(k)))$$

The guard $P(k, V0 \cup V1) = \emptyset$ may then be coded as $t(k) = 0$. As in the reachability problem discussed in the preceding issue, we represent set $V1$ by two variables $v1$ and $nv1$. Of set $V0$ only the cardinality plays a role, viz. in the assignment to ac. We record the number of elements of $V0$ in an integer variable $nv0$.

The initialization of t, $v1$, $nv1$, and $nv0$ may be coded as follows.

```
INIT:  |[j, i: int
       ; j := 0; do j ≠ N → t: (j) = 0; j := j + 1 od
       ; i := 0; do i ≠ M → t: (e(i)) = t(e(i)) + 1; i := i + 1 od
       ; j, nv0, nv1 := 0, 0, 0
       ; do j ≠ N
         → if t(j) = 0 → v1: (nv1) = j; nv1 := nv1 + 1
           □ t(j) ≥ 1 → nv0 := nv0 + 1
           fi
         ; j := j + 1
         od
       ]|
```

(The assignments to $nv0$ may, of course, be replaced by a single statement $nv0 := N - nv1$ after the repetition.)

Our solution for S then becomes

```
S:     |[nv0, nv1: int
       ; v1, t(j: 0 ≤ j < N): array of int
       ; INIT
       ; do nv1 ≠ 0
         → |[j, i: int
           ; j := v1(nv1 - 1); nv1 := nv1 - 1
           ; i := b(j)
           ; do i ≠ b(j + 1)
             → |[k: int; k := e(i); t: (k) = t(k) - 1
               ; if t(k) = 0 → v1: (nv1) = k; nv0, nv1 := nv0 - 1, nv1 + 1
                 □ t(k) ≥ 1 → skip
                 fi
               ]|
             ; i := i + 1
             od
           ]|
         od
       ; ac := (nv0 = 0)
       ]|
```

Solution of Exercise 12 (a longest path in an acyclic graph)

We have to find a statement list S such that

```
|[N, M: int {N ≥ 1 ∧ M ≥ 0}
; b(j: 0 ≤ j ≤ N), e(i: 0 ≤ i < M): array of int
    {SUC(G, b, e) ∧ G acyclic}
```

```
; |[l: int
  ; p(j: 0≤j<N): array of int
  ; S
    {p(j: 0≤j<l) records, in order, the vertices of some
     longest path in G}
  ]|
]|
```

In an acyclic directed graph there is for every vertex a maximum for the lengths of the paths ending in that vertex. Call that maximum length the *depth* of the vertex. Consider again the solution of Exercise 11. If graph G is acyclic every vertex of G is deleted from $V1$ exactly once. The order of deletion is, as we have coded it, last-in first-out. By adopting a first-in first-out order, however, the vertices are deleted from $V1$ in the order of ascending depths. In that case the vertex last deleted from $V1$ is the last vertex of a longest path. There remain thus two problems: how do we, given that last vertex, reconstruct a longest path, and, secondly, how do we code that first-in first-out order?

A longest path may be reconstructed by recording for each vertex with positive depth a predecessor whose depth is one less. We introduce an auxiliary array $pred(j: 0 \leq j < N)$ and maintain the invariant

$$(\mathbf{A}j: j \in V1 \cup V2 \wedge P(j) = \emptyset: pred(j) = -1)$$
$$\wedge\ (\mathbf{A}j: j \in V1 \cup V2 \wedge P(j) \neq \emptyset: pred(j) \text{ is an element of } P(j) \text{ whose depth is 1 less than that of } j)$$

in which $V2$ denotes the set of vertices that are not in $V0 \cup V1$. Notice that the above invariant provides sufficient information to reconstruct, given a last vertex, a longest path in G if all vertices of G are in $V1 \cup V2$. We, therefore, choose $|V1 \cup V2| \neq N$ as the guard of the repetition.

Set $V1$ is represented as a segment of array $v1$. By performing deletions at one end and additions at the other end of the segment a first-in first-out order is realized. This requires two integer variables to represent the segment. One of these, $nv2$, records the cardinality of $V2$, the other one, $nv1$, the cardinality of $V1 \cup V2$: the vertices in $V1$ are listed in the segment

$$v1(j: nv2 \leq j < nv1)$$

We have now collected all ingredients to convert the solution of Exercise 11 into one for this exercise. Since the acyclicity of G need not be tested variable $nv0$ may be dropped. We add to the initialization the setting of $pred(j)$ for all $j \in V1$:

```
INIT: |[j, i: int
      ; j := 0; do j ≠ N → t: (j) = 0; j := j+1 od
      ; i := 0; do i ≠ M → t: (e(i)) = t(e(i))+1; i := i+1 od
      ; j, nv2, nv1 := 0, 0, 0
      ; do j ≠ N
```

```
    → if t(j) = 0 → v1: (nv1) = j; nv1 := nv1 + 1; pred: (j) = −1
      □ t(j) ≥ 1 → skip
      fi
    ; j := j + 1
    od
  ]|
```

Our solution is then

```
S:  |[nv2, nv1: int
    ; v1, t, pred(j: 0 ≤ j < N): array of int
    ; INIT
    ; do nv1 ≠ N
      → |[j, i: int
        ; j := v1(nv2); nv2 := nv2 + 1
        ; i := b(j)
        ; do i ≠ b(j + 1)
          → |[k: int; k := e(i); t: (k) = t(k) − 1
            ; if t(k) = 0 → v1: (nv1) = k; nv1 := nv1 + 1; pred: (k) = j
              □ t(k) ≥ 1 → skip
              fi
            ]|
          ; i := i + 1
          od
        ]|
      od
    ; REC
    ]|
```

In REC a longest path is recorded. It is first constructed in backward order, after which its order is reversed. In the latter part a statement p:*swap*(j,k) occurs. This statement denotes for $j \neq k$ the interchange of array elements $p(j)$ and $p(k)$. If $j = k$—which is in REC precluded by the preceding guard—p:*swap*(j, k) is equivalent to **skip**.

```
REC: |[k: int
     ; k, l := v1(N − 1), 0
     ; do k ≠ −1 → p: (l) = k; k, l := pred(k), l + 1 od
     ]|
   ; |[j, k: int
     ; j, k := 0, l − 1
     ; do j < k → p:swap(j, k); j, k := j + 1, k − 1 od
     ]|
```

The solution of this exercise has, like that of Exercise 11, a computation time that is linear in the number of vertices and arcs of G.

Solution of Exercise 13 (checking whether a graph is bipartite)

It is requested to find a statement list S such that

$$
\begin{array}{l}
|[N, M: \mathit{int}\ \{N \geq 1 \wedge M \geq 0\} \\
; b(j: 0 \leq j \leq N), e(i: 0 \leq i < M): \textbf{array of } \mathit{int} \\
\quad \{\text{NEIGH}(G, b, e) \wedge G \text{ connected}\} \\
; |[\ \mathit{bip}: \mathit{bool} \\
\quad ; S \\
\quad \{\mathit{bip} \equiv (G \text{ is bipartite})\} \\
\quad]| \\
]|
\end{array}
$$

Also this program we solve by the tripartitioning technique. Its solution is actually a very good illustration of this technique. We distinguish white vertices (set $V0$), grey vertices (set $V1$), and black vertices (set $V2$). This partitioning of the set of vertices is such that no vertex in $V2$ has a neighbour in $V0$. We have furthermore that $V1 \cup V2 \neq \emptyset$ and that $SG(V2)$, the subgraph of G induced by $V2$, is connected. As a consequence, since G is connected, the condition $V1 = \emptyset$ implies that $V2$ equals the set of vertices of G.

For j a vertex, $N(j)$ denotes the set of vertices that are neighbours of j. Notice that $k \in N(j)$ is equivalent to $j \in N(k)$ and that $j \notin N(j)$. Throughout the computation $SG(V2)$ will be bipartite, i.e. $V2$ will have a subset that contains exactly one vertex of every edge in $SG(V2)$. Since $SG(V2)$ is connected, the partitioning of $V2$ into such a subset and its complement is unique. It is recorded in a boolean array $a(j: 0 \leq j < N)$, i.e. we maintain as an invariant

$$P0: \qquad (\mathbf{A}j, k: j \in V2 \wedge k \in V2 \wedge j \in N(k): a(j) \equiv \neg a(k))$$

Also for each $j \in V1$ we record a value $a(j)$. These values satisfy the following invariant:

$$P1: \qquad (\mathbf{A}j: j \in V1: (\mathbf{E}k: k \in V2 \wedge j \in N(k): a(j) \equiv \neg a(k)))$$

In order to be able to initialize $P1$ with one vertex in $V1$ and all others in $V0$, we assume the existence of an $(N+1)$st vertex that is in $V2$ and that has one vertex of G as its only neighbour. The addition of this vertex does not affect the bipartiteness of the graph. (The choice of the vertex of G to which it is connected is immaterial; we choose 0.)

In *bip* we record whether for each $j \in V1$ all neighbours $k \in V2$ satisfy $a(j) \equiv \neg a(k)$:

$$P2: \qquad \mathit{bip} \equiv (\mathbf{A}j, k: j \in V1 \wedge k \in V2 \wedge j \in N(k): a(j) \equiv \neg a(k))$$

If $V1 = \emptyset$ we conclude from $P2$ that *bip* holds. Moreover, by $P0$ and the fact that then all vertices are in $V2$, we conclude that G is bipartite. If $\neg \mathit{bip}$ holds we conclude from $P2$ that there exists a $j \in V1$ that has a neighbour $k \in V2$ with $a(j) \equiv a(k)$.

This combined with $P1$ shows that in this case graph $SG(V1 \cup V2)$, and consequently G, is not bipartite (j cannot be put in either of the two subsets into which $V2$ is partitioned). Consequently, given $P0 \wedge P1 \wedge P2$ we conclude from $V1 = \emptyset \vee \neg bip$ that

$$bip \equiv (G \text{ is bipartite})$$

holds. We, therefore, choose $V1 \neq \emptyset \wedge bip$ as the guard of the repetition.

In the repetition we move one vertex j from $V1$ to $V2$. Since $P2 \wedge bip$ holds at each step of the repetition, this move maintains $P0$. All $k \in N(j) \cap V0$ are moved to $V1$ and, in order to maintain $P1$, receive the appropriate a-values. $P2$ requires that all $k \in N(j) \cap V1$ are checked for their a-values. The program will, consequently, have the following structure:

$V0 :=$ *the set of all vertices except* 0
; $V1, V2 := \{0\}, \emptyset$
; $a: (0) =$ *true*; $bip :=$ *true*
; **do** $V1 \neq \emptyset \wedge bip$
  → *let* $j \in V1$
  ; $V1, V2 := V1 \setminus \{j\}, V2 \cup \{j\}$
  ; *for all* $k \in N(j)$ *while* bip:
    if $k \in V0 \rightarrow V0, V1 := V0 \setminus \{k\}, V1 \cup \{k\}; a: (k) = \neg a(j)$
    □ $k \in V1 \rightarrow bip := (a(k) \equiv \neg a(j))$
    □ $k \in V2 \rightarrow$ **skip**
    fi
od

The value of $a(0)$ is immaterial: $a: (0) =$ *false* would have done as well.

Again the program has the well-known structure associated with the tripartitioning technique. Initially one vertex is grey and all others are white. Per step of the repetition one grey vertex is coloured black. This requires its white neighbours to be coloured grey and the invariants to be reestablished. The latter involves a case analysis: one case for each colour a neighbour of the vertex selected can have.

In order to be able to express the guards in the selection, we introduce an integer array $v(j: 0 \leq j < N)$ and maintain

$$\begin{aligned}(\mathbf{A}j: 0 \leq j \leq N: \quad & (v(j) = 0) \equiv (j \in V0) \\ & \wedge (v(j) = 1) \equiv (j \in V1) \\ & \wedge (v(j) = 2) \equiv (j \in V2))\end{aligned}$$

This representation does not suffice to choose a vertex $j \in V1$ readily. We, therefore, represent $V1$ in the usual way as well: the vertices in $V1$ are listed—in some order—in the segment

$$v1(j: 0 \leq j < nv1)$$

Thus we come at the following program text:

```
S:      |[nv1: int
        ; v, v1(j: 0≤j<N): array of int
        ; a(j: 0≤j<N): array of bool
        ; v: (0) = 1
        ; |[j: int
          ; j := 1; do j ≠ N → v: (j) = 0; j := j+1 od
          ]|
        ; nv1 := 1; v1: (0) = 0
        ; a: (0) = true; bip := true
        ; do nv1 ≠ 0 ∧ bip
          → |[j, i: int
            ; j := v1(nv1 − 1); nv1 := nv1 − 1; v: (j) = 2
            ; i := b(j)
            ; do i ≠ b(j+1) ∧ bip
              → |[k: int; k := e(i)
                ; if v(k) = 0 → v: (k) = 1; v1: (nv1) = k; nv1 := nv1 + 1
                                ; a: (k) = ¬a(j)
                  □ v(k) = 1 → bip := (a(k) ≡ ¬a(j))
                  □ v(k) = 2 → skip
                  fi
                ]|
              ; i := i+1
              od
            ]|
          od {bip ≡ (G is bipartite)}
        ]|
```

The solution is linear in the number of vertices and edges of G.